Flat Chart Technique for Embedded OS Testing

Victor V. Nikiforov and Sergey N. Baranov
St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences (SPIIRAS)
{nik,snbaranov}@iias.spb.su

Abstract. A special language to define the respective testing task logic and the concept of flat charts to efficiently perform an embedded OS execution-based testing presented. To avoid heavy interpreting of text strings during test runs, strings are pre-processed and converted into a regular array form.

Functional testing checks the correctness of the OS behavior and finds defects in:

- execution of basic OS directives invoked from tasks and ISRs;
- processor switches among threads;
- data and signal transactions;
- error handling routines.

Timing testing measures the following timing data on OS execution:

- local time – execution time of a particular OS directive;
- global time – total execution time of the whole application;
- latency – time interval between the moment when the interrupt occurred and the moment when the interrupt started to be processed by a respective ISR.

Observed Testing Rules:

- *focus* – each test checks only one OS feature under particular conditions with only two possible outcomes: pass or fail;
- *repeatability* – same behavior at each test execution;
- *non-interference* – no intrusion into OS functioning (no access to OS variables, command lines, or structures);
- *black-box approach* – no knowledge/assumptions on the OS inner structures.

The Flat Chart Technique ensures repeatability of the test execution order under the following constraints:

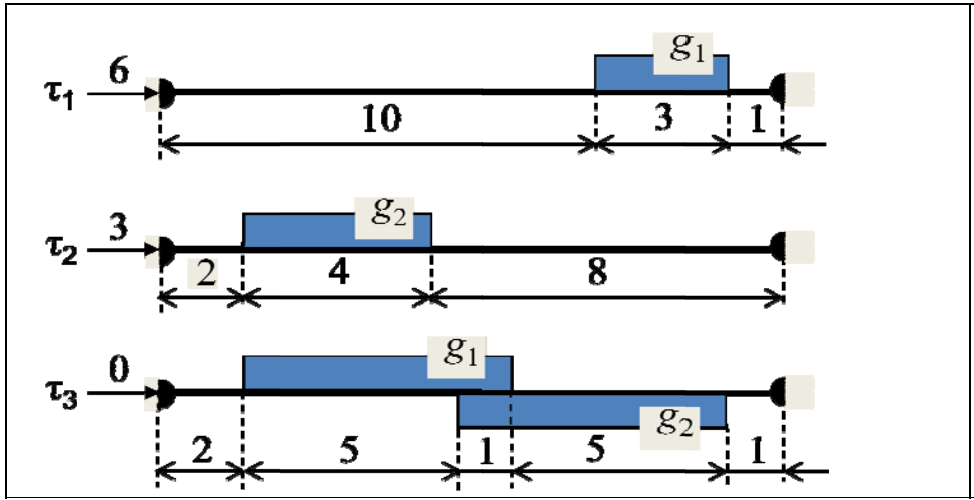
• test utilities in the test application are simple and small in number; • tests contain invocations of only OS service operators and test utilities;	• all tasks are generated statically; • task priorities are static; • no two tasks have the same priority.

The Test_Step structure may be formally defined as:

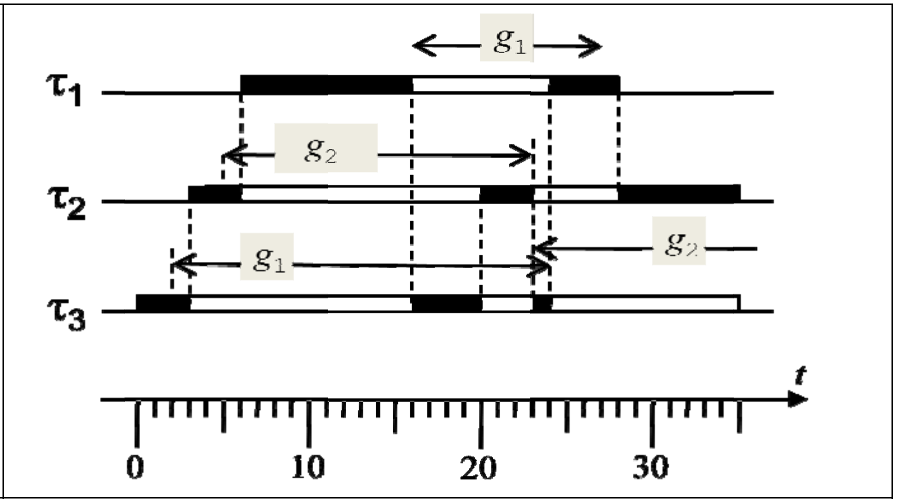
```
typedef struct Test_Step  {                union StepArg                {
    int TaskId;                                  int IntArg;
    void (* UtilServ) (  );                     char* StringArg;
    StepArg Arg_1;                              void (* FunArg) ( );
    StepArg Arg_2;    } TestStep;                         ...              };
```

Example: Three tasks τ_1, τ_2, and τ_3, sharing three global resources g_1, g_2, and g_3.

Profile of the `ThreeTasks` Application



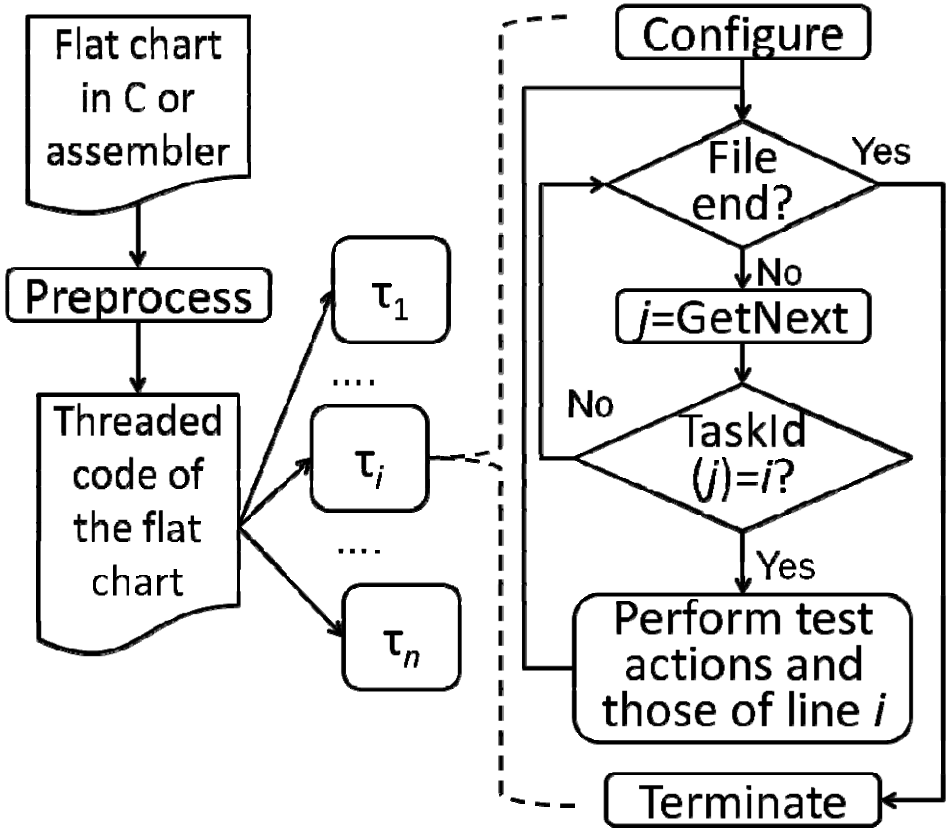
The Task Execution Order



Flat chart: Rows specify the execution order, columns reflect the task priorities

```
          1       2        3
TestStep  ThreeTasks =       {
  1)                       3, &ActivateTask, 3, 0
  2)                       3, &Lock, 1, 0
  3)              2, &ActivateTask, 2, 0
  4)              2, &Lock, 2, 0
  5)      1, &ActivateTask, 1, 0
  6)      1, &Lock, 1, 0
  7)                       3, &Lock, 2, 0
  8)              2, &Unlock, 2, 0
  9)                       3, &Unlock, 1, 0
 10)      1, &Lock, 1, 0
 11)      1, &EndTask, 1, 0       }
```

The workflow of the test interpreter



Developing Scenario Tests with Flat Charts

Each scenario test realizes a sequence of actions based on some underlying idea.

A flat chart to check a message exchange mechanism which provides message pointer passing from an ISR to a task or from one task to another. The variables *mes1_ptr*, *mes2_ptr*, and *mes3_ptr* are message pointers.

```
        1      2      3
TestStep MsgTravel [  ] =          {
 1)   1,&CallGetMsg, &mes1_ptr,  0                 // No msg, TASK_1 is waiting
 2)         2,&ResumeIsr,  0,  0                   //  Interrupt is  simulated
 3)  -1,&CallPutMsg, TASK_3, TEST_MSG              // Send msg to TASK_3
 4)         2,&CallGetMsg,  &mes2_ptr,  0          //  No msg, TASK_2 is waiting
 5)               3,&CallGetMsg,  &mes3_ptr,  0    //  TASK_3 received Msg
 6)               3,&CallPutMsg,  TASK_1, &mes3_ptr //  Activate TASK_1
 7)   1,&CallPutMsg,  TASK_2, &mes1_ptr            // TASK_2 becomes ready
 8)   1,&CallTaskEnd,   0,  0                      //  Activate TASK_2
 9)         2,&Check_Equal,  &mes2_ptr, TEST_MSG   //  Is msg the same
10)         2,&End_of_Test,   0,   0               // as TEST_MSG?
11)   0,&End_of_scheme,  0, 0   }
```

This scenario of message passing between tasks consists of the following events:

1) *Task_1* tries to get a message & becomes suspended as there's no message yet;

2-3) ISR passes the message *TEST_MSG* to *Task_3* not ready yet to receive it;

4) *Task_2* tries to receive a message which is absent and becomes suspended;
5-6) *Task_3* receives the message *TEST_MSG* sent previously by the ISR and resends it to the suspended *Task_1* which was waiting for it;
7-8) *Task_1* resends the message to *Task_2* and frees the processor through invoking the service procedure *TaskEnd*();
9) Upon termination of *Task_1* the message received by *Task_2* is compared to *TEST_MSG* – the two message pointers should coincide.
10-11) Termination of the test run.

Loops in Flat Charts: Two New Elements &LoopStart and &LoopEnd

```
        1      2
//   -----    Flat chart with a single loop   ------
TestStep MessQueue [  ] =         {
// ----  The message queue with 10 messages is formed for Task_2
1)    0, &LoopStart, &cycle_var, 10
2)    1,&CallPutMessage, Task_2, &mes1_ptr     // Repeat msg send
3)    0, &LoopEnd, &cycle_var, 0
4)    1, &CallTaskEnd, 0, 0                    // Task_1 terminates
 // ---- The message queue of 10 messages is consumed by Task_2
5)    0, &LoopStart, &cycle_var, 10
6)           2,&CallGetMessage, &mes2_ptr, 0   // Repeat msg receive
7)    0, &LoopEnd, &cycle_var, 0
8)    0,&End_of_scheme, 0, 0    }
```

Testing the Error Handling Service with the &CheckErrData Element

```
//   -----    Flat chart for error service testing   ------
TestStep MemReqErr [  ] =        {
// .......Steps from Step_01 to Step_i exhaust all memory resource
i+1)    1,&GetMemory, 30, &mem_ptr                //  Step_i+1
        0, &CheckErrData, NO_MEMORY, 0
 // ..................... Remaining elements of the MemReqErr array
        0,&End_of_scheme, 0, 0  }
```

Task_1 requires 30 memory blocks which causes an error because the memory resource becomes exhausted. An error shall invoke a special thread of actions which enters the flat scheme interpreter body. The interpreter finds the respective auxiliary line in the flat chart and performs the *CheckErrData*() utility assuming that the OS reports the *NO_MEMORY* error code into the error handling block.

Automated Test-Run Sessions: Local/Global Time Measurement

The precision of OS time service directives is not appropriate for local time measurements; therefore, direct access to the hardware time register is needed. Test actions for local time measurement are: (A) read the timer register (take the time *before* the action); (B) call the *UtilServ*() utility for executing the specified action; (C) read the timer register again (take the time *after* the action); (D)store the results of measurement and perform log operations with these data.

Global time measurements are less precise; therefore, the OS time service may used:

```
        1       2
TestStep HighPriorTaskSwitching [  ] =   {
1)      1,&CallSysTime, &start_time, 0              // Store the start time
2)      0, &LoopStart, &cycle_var, 1000             // Initialize the loop
//  ------ The set of operations for measurements  ------
3)      1,&CallGetMessage, &mes1_ptr, 0             //  Suspend Task_1
4)              2,&CallPutMessage, &mes1_ptr, 0// Send msg to Task_1
5)      0, &LoopEnd, &cycle_var, 0                  // Terminate loop operations
6)      1,&CallSysTime, &finish_time, 0             // Store the end time
7)      0,&LogGlobalTime, 0, 0                      // Store the result
8)      0,&End_of_scheme, 0, 0          }           // End of scheme
```

The number N of measurement cycles depends on the relation between the precision ΔT_m of the *SysTime*() mechanism, duration T_c of one application cycle, and duration T_p of the *LoopStart*() and *LoopEnd*() operations (assuming they are equal). The larger the value of $N \times (T_c/(\Delta T_m+T_p))$, the more precise measurement results will be obtained.

Thus, the flat chart technique provides a convenient form for understanding and describing tests for global time measurement.

Latency Testing. The simplest statistical way of latency measurement assumes simultaneous execution of two logically isolated components: (A) a benchmark application with a set of interacting tasks; (B) a dedicated ISR to calculate difference between moments of the measurement interrupt and of the start of its processing.

With this approach, a single result L_m of measuring the latency value will be less than or equal to the maximal possible latency value L_r. The difference $d=L_r–L_m$ represents the inaccuracy of result of a single latency measurement.

To achieve higher accuracy of latency measurement, single measurements are performed n times and the maximum of L_m is considered as the final result. The required accuracy of the final result is achieved with the probability not less than $1–(1–\Delta t/T)^n$.

Measuring Code Coverage. The technique is based on using codes of prohibited TRAP instructions and is realized with another designated vector of TRAP-interrupts.

Enhancements of the Flat Chart Technique. For real asynchronous action threads (as required for latency measurements), methods beyond the flat chart scheme should be used. The flat chart technique may be further extended to distributed OS testing. In this case, a test application is a program with real parallelism and if quasi-asynchronous execution turns out to be suitable for particular testing, then the only needed extension is refinement of action flows naming. Otherwise, a separate flat chart should be developed for each physical processor with additional means for cross-referencing.

Conclusions. The flat chart technique gives an efficient way to develop test suites for embedded OS execution-based testing with well-structured and understandable descriptions of the test applications with tasks and ISRs for parallel execution. It allows to check the correctness of implementation of basic OS mechanisms – data and signal exchange among action threads, run-time allocations of memory, special structures, and processor’s time and to perform measurements of local/global time, OS latency, and code coverage.

References

[1] Li Q., Yao C. Real-time concepts for embedded systems. CRC Press (2003).

[2] Thane H., Hansson H. Testing distributed real-time systems. Microprocessors and Microsystems 24(9), 463–478 (2001).

[3] Desikan S. Software testing: principles and practice. Pearson Education India (2006).

[4] Myers G.J., Sandler C., Badgett T. The art of software testing. 3nd edn. John Wiley & Sons, New York (2011).

[5] Hailpern B., Santhanam P. Software debugging, testing, and verification. IBM Systems Journal 41(1), 4–12 (2002).

[6] Brodie L. Thinking Forth. Punchy Pub (2004).

[7] Biswal B. N. Pragyan N., Durga P. M. A novel approach for scenario-based test case generation. In: International Conference on Information Technology 2008 (ICIT'08). IEEE, (2008).

[8] Lefticaru R., Florentin I. Automatic state-based test generation using genetic algorithms. In: International Symposium on Symbolic and Numeric Algorithms for Scientific Computing 2007 (SYNASC). 178–195. IEEE (2007).